HARDWARE-ASSISTED SECURITY: BLOOM CACHE – SCALABLE LOW-OVERHEAD CONTROL FLOW INTEGRITY CHECKING

A Thesis
Presented to
The Academic Faculty

by

Vinson Young

In Partial Fulfillment
of the Requirements for the Degree
Master's in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
August 2014

HARDWARE-ASSISTED SECURITY: BLOOM CACHE – SCALABLE LOW-OVERHEAD CONTROL FLOW INTEGRITY CHECKING

Approved by:

Dr. Jongman Kim, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. John Copeland
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: April 25th 2014

ACKNOWLEDGEMENTS

I would like to thank my adviser Jongman Kim for advising my research direction. I would also like to thank my family, without whose guidance and support I would not be here.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

IDS	Intrusion Detection System
CFI	Control Flow Integrity
DEP	Data Execution Prevention
ASLR	Address Space Layout Randomization
CFG	Control Flow Graph
RAS	Return Address Stack

SUMMARY

Computers were not built with security in mind. As such, security has and still often takes a back seat to performance. However, in an era where there is so much sensitive data being stored, with cloud storage and huge customer databases, much has to be done to keep this data safe from intruders.

Control flow hijacking attacks, stemming from a basic code injection attack to return-into-libc and other code re-use attacks, are among the most dangerous attacks. Currently available solutions, like Data execution prevention that can prevent a user from executing writable pages to prevent code injection attacks, do not have an efficient solution for protecting against code re-use attacks, which can execute valid code in a malicious order.

To protect against control flow hijacking attacks, this work proposes architecture to make Control Flow Integrity, a solution that proposes to validate control flow against pre-computed control flow graph, practical. Current implementations of Control Flow Integrity have problems with code modularity, performance, or scalability, so I propose Dynamic Bloom Cache, a blocked-Bloom-filter-based approach, to solve current implementation issues.

CHAPTER 1

INTRODUCTION

Computers were not built with security in mind. As such, security has and still often takes a back seat to performance. However, in an era where there is so much sensitive data being stored, with cloud storage and huge customer databases, much has to be done to keep this data safe from intruders.

Many solutions are available on the market--network solutions do a good job of filtering and finding many intrusions. However, they are not infallible. When network solutions fail, you need a good host-based IDS (intrusion detection system). Signature and anomaly-based solutions are the norm currently, with some compartmentalization and low-level memory-based security solutions available. For situations where security is of utmost importance, though, heuristic solutions may not be sufficient--we need a strong low-level protection solution to make a system provably secure. On the other hand, one must not throw aside performance and compatibility in the effort of making a secure system.

I argue that a strong memory-level protection, in particular, the control flow integrity (CFI)[1] paradigm, should be used to secure a system, and I offer a solution that addresses the shortcomings of current proposals in terms of scalability, modularity, and overhead.

Motivation & History

Computer systems are constantly under attack, and as security improves, so do the attacks. To understand the vulnerabilities, one must first go through the types of attacks available to understand the vulnerabilities we aim to protect. In this case, we aim to protect against control flow hijacking attacks.

One of the earliest types of attack techniques is a code injection attack. This technique usually involves use of buffer overflow to get the program to execute code on the stack. This type is largely protected against by hardware W(+)X or Data Execution Protection [2] techniques that stop code from being both writable and executable.

Instead, we turn our attention to code re-use attacks. To circumvent W(+)X mechanisms, code re-use attacks use code regions that have already been cleared to run. The earliest type, called return-into-libc (RILC) [3] attacks, involves changing a return address to return to linked library code in an unexpected way for an attack. RILC can involve calling system calls to create a writable and executable memory region to perform previous attacks. More expressive attacks, called Return-Oriented Programming [4] / Jump-Oriented Programming [5], base their attacks on the idea that code fragments ending in returns or jumps can be combined to form a Turing-complete programming method.

Currently in-use protection mechanisms of Data Execution Prevention and Address Space Layout Randomization [6] are efficient, but do not fully protect the user. DEP can be countered by code re-use attacks while ASLR can be defeated [7] [8], so there is a need for a stronger protection against control flow hijacking attacks.

Proposed Solution – Practical Control Flow Integrity

Control Flow Integrity (CFI)[1], proposed by Abadi, et al. is a powerful security paradigm for protecting against generic control flow attacks. Control Flow Integrity works by verifying the branches of a program by comparing the current branch to branches in a pre-computed Control Flow Graph (CFG). The theory proposed by CFI is strong as it prevents control flow hijacking at a very basic level—by restricting the number of valid branches to a small set. However, the implementations proposed have all come at a cost of a combination of problems, including some set of binary compatibility, performance, or scalability, and, as a result, CFI has not found widespread adoption.

The main issues of CFI techniques can be shown by observing the dichotomy of CFI techniques--many papers propose using an inlined-CFG and many others propose to use a meta-CFG. Inlined CFG's, as in the original CFI paper by Abadi, et al., offer relatively low performance overhead as the CFG has the same locality as the instruction stream. This comes at the cost of binary rewriting, which increases binary size and, more importantly, decreases compatibility of libraries as the offsets are changed. Techniques based on meta-CFG's, on the other hand, require no binary rewriting but are often plagued with large performance overhead due to the explicit tracking of state and context switching needed to perform the checks.

I propose to use a Bloom-filter-based implementation [9] to overcome the implementation issues of the previous dichotomy of CFI, and I further improve upon it to introduce scalability and locality. A Bloom-filter-based design that stores branches into a single Bloom filter could obviate the use of the binary rewriting that causes modularity issues while simultaneously provide a stateless way to check branches efficiently. However, when one considers that all relevant branches / branch signatures must fit within this single Bloom filter structure, concerns about the scalability of this approach come up.

Y. Shi and G. Lee have previously proposed a single stateless 32kB Bloom filter design to verify (source || destination || "execution path") branch behaviors [10]. Given that a recently added uop cache in Intel's Sandy Bridge architecture used ~6kB and required simplification of other branch prediction logic to fit it [11], I argue the space requirements of the previous Bloom filter proposal are both too large space-wise, and worse yet, not large enough as "gcc" could not fit in this Bloom filter structure[10]. I believe this scalability issue is what prevents adoption of a stateless control flow integrity solution, and offer a solution in this work.

I propose a 2kB blocked Bloom filter, called Bloom Cache, which caches smaller 64B blocked Bloom filters to reduce the cache space requirements, with new methods to

maintain spatial and temporal locality to the proposed structure to reduce its pressure on memory. The proposed methods include a new way to make use of dynamic locality by drawing from compressed cache architecture ideas and a new way to organize blocked Bloom filters to increase spatial locality.

The main contribution of this work is in solving the debilitating scalability and compatibility issues of current Control Flow Integrity implementations.

New contributions in this work include:

0. Blocked-Bloom-filter-based approach to overcome scalability and modularity issues from previous approaches

1. A method to maintain spatial locality in a blocked Bloom filter design

2. A dynamic / static method to add temporal locality in an unbalanced blocked Bloom filter design

3. New method to add the ability to trade security and performance dynamically

4. Analysis of additional methods of dynamic signatures

CHAPTER 2

BACKGROUND

There are a wide variety of security solutions available, including various types of network-based security and host-based security. To achieve a suitable level of security efficiently, network administrators and end hosts usually employ security at multiple levels.

General / Network Security

In terms of network solutions available, firewall, port checking, intrusion detection systems (IDS), and encryption are widely used [12]. Firewalls and port scanning for vulnerabilities can be used to protect against exploits on ports, and, when integrated with application-level or host-based IDS, firewalls can also protect applications. With respect to IDS, signature-based, anomaly-based, and flow-based are the most common. Signature-based solutions work by finding a signature of a previously seen attack and verifying against this signature. Signature-based solutions are effective against previously-seen attacks but are unable to detect new attacks. Anomaly-based solutions work by classifying normal-use patterns and generating warnings when behavior deviates from the supposed behavior. Anomaly-based solutions can be used to protect against zero-day attacks, but this comes at a cost of higher false positive rate and can miss attacks that have normal traffic patterns. Flow-based solutions keep track of the state of connections and sessions, and, using this information, flow-based solutions can also detect anomalous behavior. Encryption in general also provides confidentiality and can also provide integrity.

The scope of this paper, however, largely deals with end host security, which is important when aberrant behavior is not caught by the previous solutions.

Host-based Security

Compartmentalization

In addition to the behavior-based solutions of before, host-based solutions often comprise of two categories: compartmentalization and memory-level protection. Compartmentalization is useful in keeping attacks localized from integral resources while memory-level protection more directly monitors memory access to protect memory integrity and control flow.

Compartmentalization is often used as a general security measure and exists in many forms—rule-based policies and virtualization being very popular. Rule-based policies offer an OS-level protection in general by restricting access to files and transactions with permissions, and this level of protection is commonly seen in the form of Role-based Access Control [13], Mandatory Access Control, Discretionary Access Control, Bell-LaPadula model, Clark-Wilson, and the other similar policies. Virtualization policies are also often proposed due to virtual machine's inherent separation of resources. These approaches include the recent TrustZone [14] and Hypervisor-based [15] capabilities. Trustzone works by having a secure zone and an insecure zone with kernel and user level capabilities. Its API ferries instructions between the zones and manages to keep important data secure. Hypervisor-based, like Hypersafe [16], concepts work by keeping the actual kernel small and protected while virtualizing other operating systems so the attacks are localized to the virtualized operating system. There are various other ideas, such as HyperCoffer [17] and HyperCheck [18], that improve upon such concepts, but the concept is generally similar.

Memory-level Protection

Another general approach is protecting against memory attacks themselves[19]. There are many levels of memory an attacker can mount an attack on, and most of such

memory attacks can be categorized into code corruption attacks, control-flow hijacking attacks, data-only attacks, and information leaks. There are a multitude of solutions against the latter two types of attacks, such as memory safety, data integrity, and data-flow integrity [20]. Such concepts are interesting in theory, but sadly, not much deployment has been seen in these fields as their overheads are higher than industry can accept. The story, however, is different for control-flow hijacking. There has been some active deployment, in the form of Data Execution Protection [2], Address Space Layout Randomization[6], and other compiler-based techniques, as protection on this layer is more efficient, making control flow protection a promising direction to follow.

Data Execution Protection (also called W(+)X protection) [2], introduced to Linux and Windows in 2004, protects on a hardware level by making sure a page is not both writable and executable at the same time. Making sure a page is not executable and writable at the same time defeats many code injection attacks in that DEP can make the stack not executable. DEP is a very useful first step in maintaining code integrity, but it does not catch all cases as it is vulnerable to code re-use attacks, which uses code that has previously been approved as valid executable code. Compiler improvements have bolstered the defenses against most return-into-libc attacks, but there are still code re-use attacks that have not been properly defended against

Address Space Layout Randomization [6] is also commonly used in commodity hardware. ASLR is based on the premise that randomizing locations of libraries and important information makes it more difficult for an attacker to launch an effective attack. There are exploits to ASLR [7][8], but due to its low overhead and widespread integration, it is used in most systems currently.

Recent literature has also proposed new ideas to protect stack and memory. Dynamic Information Flow Tainting [21] works by marking, or "tainting," potentially insecure instructions and keeping track of what the "tainted" instruction has accessed. This method is efficient and can protect a system if tuned correctly per application, but it

has a downside that false positives could propagate the taint unnecessarily. Another proposed mechanism works by finding kernel-level or other program invariants and making sure those values are not changed [22]. These proposals are interesting, but they may not provide full protection as they assume an attack that modifies the kernel. Others yet provide stack protection [23] and pointer protection [24], which provide some bounds checking to counter buffer overflows.

Additionally, Control Flow Integrity is a promising idea that lies in the category of memory defense. Control Flow Integrity works by verifying all branches against a pre-computed control flow graph. Current proposed implementations have not seen industry use due to problems with compatibility and performance overhead. As such, there is room for improvement.

Bloom Filters

A Bloom filter is a space-efficient structure with constant access times that can quickly tell when an element is part of a set of elements stored in the Bloom filter. A Bloom filter is similar to a hash structure and in general works by having multiple independent hashes that can set or check corresponding bits in a block of memory. Starting with an empty block of memory initialized to all 0's, insertion works by setting, for example, 3 bits as 1's.To check whether an element is in the Bloom filter, all one has to do is hash to find the 3 corresponding bits in the example and check if they are all 1's. This method allows constant space usage for any number and size of elements. However, as more elements are inserted, the possibility of a new element being decided that it is already part of the set increases as more of the bits are set to 1. Thus, Bloom filters allow one to trade a small false positive rate for a lower amount of space used [25]. There are many variations on this basic Bloom filter, and this work in general is loosely based on a blocked Bloom filter design.

A variation I make use of in this work is called a blocked Bloom filter. In the blocked Bloom filter, a single large Bloom filter is divided into smaller Bloom filter "blocks." Membership testing is done by first indexing into the right "block" and subsequently performing a normal Bloom filter membership check on that smaller "block." This allows for better cache performance by making localizing the N-bit comparisons in a bloom filter check to a single cacheline. This provides improved accesses in hardware, but the hashing to decide which block to index into is still random and thus could be improved. I make use of this organization by further introducing locality into the indexing stage so that block accesses are also local.

CHAPTER 3

LITERATURE REVIEW OF CFI IMPLEMENTATIONS

Control Flow Integrity offers a strong general control-flow hijacking protection, but current implementations have not seen widespread use due to various weaknesses existing within each implementation. Current implementations can be widely categorized as being either primarily hardware or primarily software-based. Within those solutions, they can additionally be subdivided into inlined and meta-CFG approaches.

CFI – Software-based – inline vs. meta-CFG

The original CFI paper [1] proposed a software method of inlining checking code and verification tags into the binary directly. This proposal was effective in that it had offered a way to implement CFI purely with software and with tags that had the same locality as the code. As a reference, the performance overhead was a modest 16% on average. A potential reason why this approach has not seen widespread use is that it employs binary rewriting, which increases binary size and changes function offsets. Changing the offsets upsets code modularity/compatibility by changing the addresses external functions must call. Additionally, the tags are "imprecise" in that function calls that have the same destination must be part of the same group, making the protection weaker. There are many other proposals that employ binary rewriting to perform checks [26] [27].

An alternative method to using code rewriting is to compare against a meta-CFG, a control flow graph that is separate from the source code. These papers mostly use some interrupt on indirect branches to check if the branches match a pre-computed CFG [28]. These papers are in general plagued with overhead caused by the excessive context switching necessary for the checks and often use potentially large CFG files. There are

optimizations in terms of less frequent checks [28] and more local storing methods to increase locality, but performance overhead is generally high.

CFI -- Hardware-based

Branch Regulation [4] is a recent proposal that annotates function bounds into the binary to compare for validity of branches. Branch Regulation introduces a small comparator unit into the pipeline to check if branches are within bounds and finds the bounds by looking up either in software or a hardware cache of recent function bounds. This is an approximation of CFI that looks promising.

Shi [10] proposes an interesting way to implement CFI. It proposes to store the (source||destination||branch history) behavior signatures into a hardware Bloom filter of a set size and checks branches by testing if the (src||dst||bh) tuple is part of the set. Employing a Bloom filter to store valid patterns has the properties of being space-efficient and having good access times despite no explicit controlling of state. As this method falls under the category of meta-CFG, it also removes the binary compatibility problem introduced by binary rewriting. However, the configuration proposed comes at the cost of scalability, area, and potentially context switching inertia. As the hardware Bloom filter was designed to be a set size, branches that do not fit inside the Bloom filter must instead be checked in software. The implementation proposed uses a 32kB Bloom filter, with potentially 2 extra Bloom filters to lower false positive rates. The 32kB Bloom filter was also not able to fit gcc execution and could cause expensive software checks. In the event of a context switch, the 32kB would also likely need be flushed and re-read the next time the program receives execution time. Overall, it seems space and scalability were not sufficiently considered.

Room for Improvement

Current proposals have some room for improvement: software methods have problems with code modularity or performance and hardware implementations could be approximate or not scalable. I offer a solution based on blocked Bloom filters that solves issues of code modularity and performance from software implementations, and I improve upon it to solve issues of scalability and locality of reference from hardware implementations.

CHAPTER 4

BLOOM CACHE

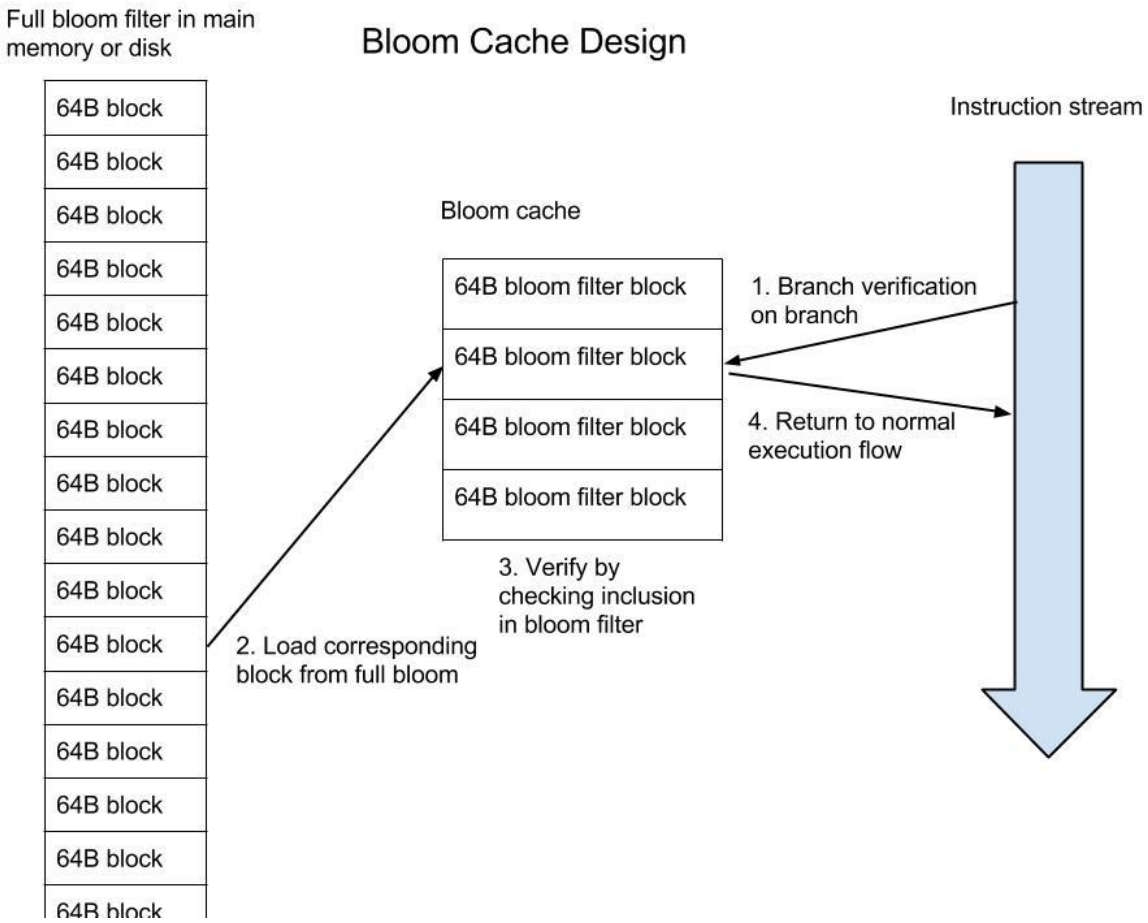

Figure 1: General Bloom Cache Design

As a high-level overview, the proposed Bloom Cache stores (src, dst) tuples into a Bloom filter structure and verifies branches seen during program execution by checking for set inclusion in the full Bloom filter structure. However, internally, the tuples are stored and accessed in a way such that it will perform well in cache regardless of the number of valid execution paths. An example configuration of a Bloom Cache would be a full Bloom filter that uses a 2kB cache composed of 32 64-Byte Bloom filter blocks to cache active branch patterns. Examples in this chapter use 64B Bloom filter blocks to store 64 branches to have a ~2% false positive rate [25].

The basic premise of the Bloom Cache is that by dividing the Bloom filter into spatially and temporally local blocks and keeping only relevant blocks in cache, one can verify branches efficiently with a minimal amount of area while maintaining performance and stateless properties gained from employing Bloom filters. To help understand the proposed Bloom Cache design, I will go through various designs to show how the proposed design addresses scalability and subsequently locality of references.

4a. Blocked Bloom provides a good starting framework.

4b+c. Static Blocked Bloom ideas provide spatial locality, but has limits.

4d. Dynamic Blocked Bloom ideas incorporate temporal locality dynamically.

Blocked Bloom Filter – Scalability and Area Requirements

A single large Bloom filter, as proposed in [10], has the property of constant access regardless of space, but it comes at a cost of flexibility and scalability as a hardware Bloom filter must be designed to a certain size. More elements can continuously be put into a Bloom filter of set size, but the false positive rate will increase until a degenerate case where every single branch that is checked is deemed valid. To alleviate the problem of scalability, I propose to use a blocked Bloom filter.

A blocked Bloom filter works by storing a Bloom filter into multiple smaller Bloom filter "blocks." To decide which block to access to perform a Bloom filter check, one first hashes a value to find the index of the Bloom filter block. This basic idea allows smaller Bloom filter blocks to be placed in and out of cache as needed.

However, using a blocked Bloom filter as is comes at a cost of locality. Since values are hashed to find which Bloom filter block to access, the access to the blocks are random--this would require the whole Bloom filter to fit into memory or cache to still be efficient. To introduce locality in the blocks to reduce random access patterns, I propose ignoring the lower order bits in the (src, dst) pairs to decide which block to access. With

this method, branches with similar spatial locality would be grouped together, reducing the number of random accesses. Introduction of spatial locality into the structure was effective at reducing the number of memory accesses the cache must make, as hit rates in subsequent configurations are easily >90%.

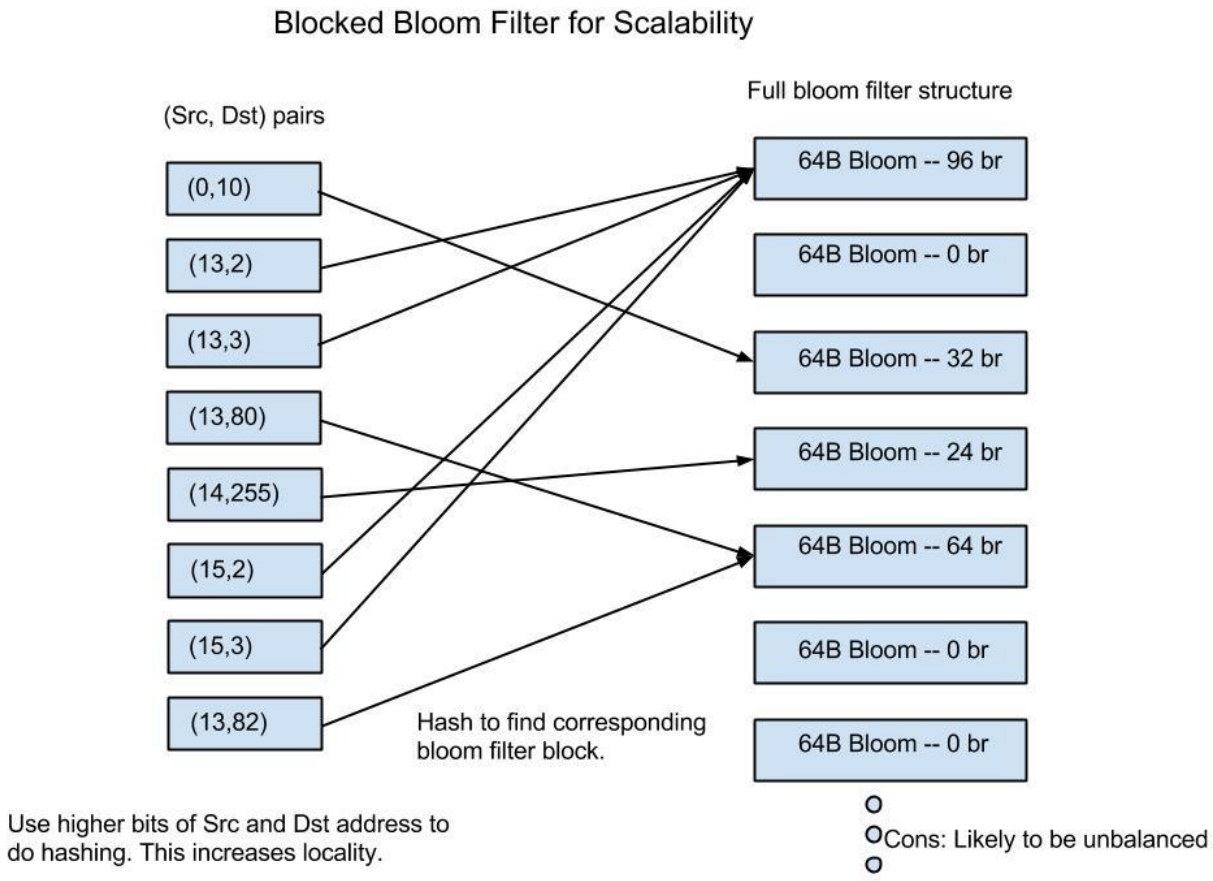

Figure 2: Blocked Bloom Filter

This method of indexing blocks increases spatial locality, but it unbalances the Bloom filter blocks because valid branches patterns are not uniformly distributed. However, this provides a starting framework that has:

1. Scalability, in that only a small portion of the Bloom filter needs to be cached at a time

2. Spatial locality of reference, in that branches are more likely to branch to spatially local instructions

3. Unbalanced blocks, as valid branch patterns are not uniformly distributed

Spatially-Ideal Static Blocked Bloom Filter

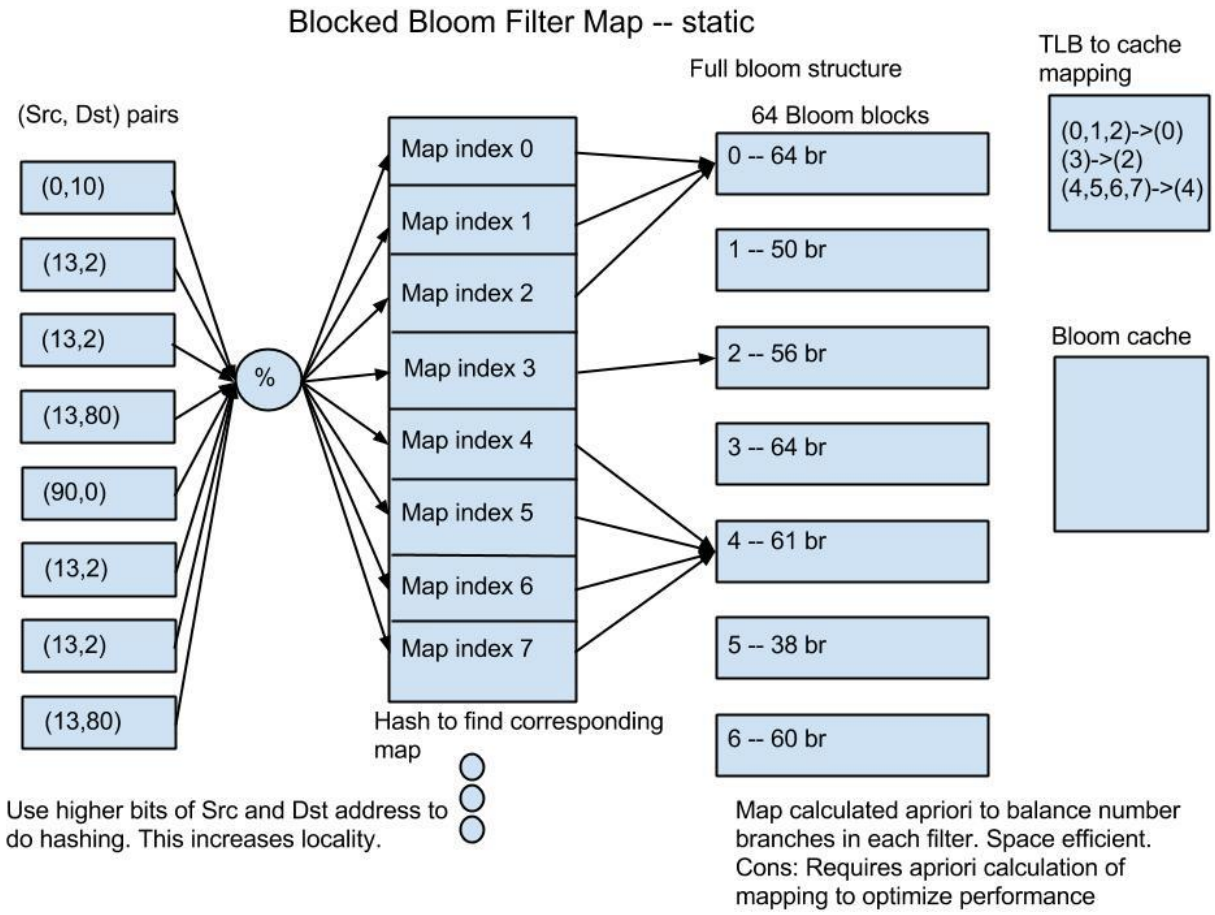

Figure 3: Static Blocked Bloom Filter

A simple solution to unbalanced blocks would be to just combine spatially local blocks. This is a static method to increase spatial locality and balance the Bloom filter blocks. As is shown in a later section (6a), blindly combining spatially local blocks initially increases performance but reduces in effectiveness as more branches are added into the structure. Thus, this organization, while it improves upon previous implementations in terms of locality, is not fully scalable. This scalablity is addressed in section (4d) by adding temporal locality.

The current implementation of this structure ignores the organization needed to find the corresponding Bloom filter blocks and is provided to show potential gains from using only spatial locality. One could potentially implement this by using a mapping structure as shown, or one could more efficiently merge local blocks by using a K-map representation. Additionally, one could combine temporally local blocks by profiling the application, but this is left for future work.

Hierarchical Bloom Filter

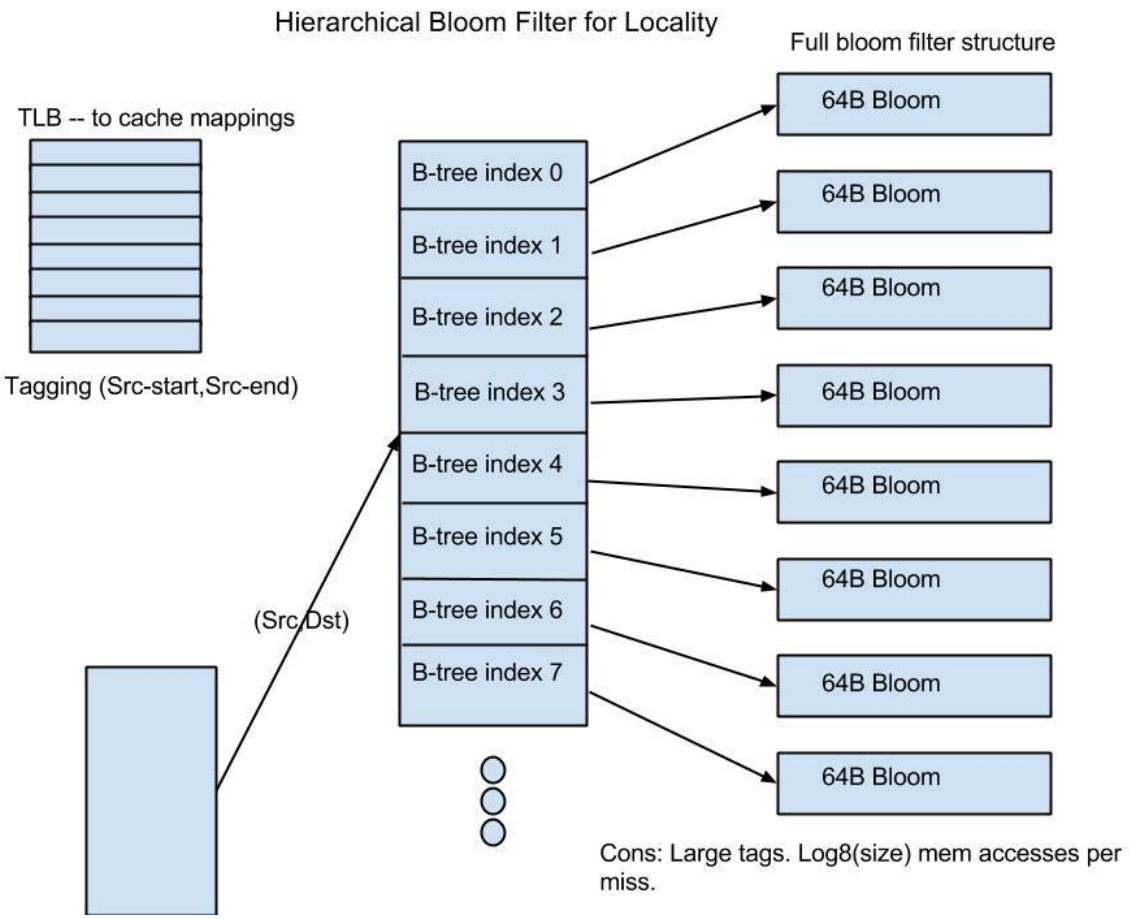

Figure 4: Hierarchical Bloom Filter

To implement the ideally local described in the previous section, one can implement a tree structure to organize the blocked Bloom filters. An 8-way btree is presented as an example. The space overhead from using this method is low, but this method requires

multiple accesses to find the corresponding Bloom entry. Additionally, as the number of branches increases, the number of accesses needed to find the corresponding Bloom entry increases.

Dynamic Bloom Cache - Dynamic Locality and Configurable Security

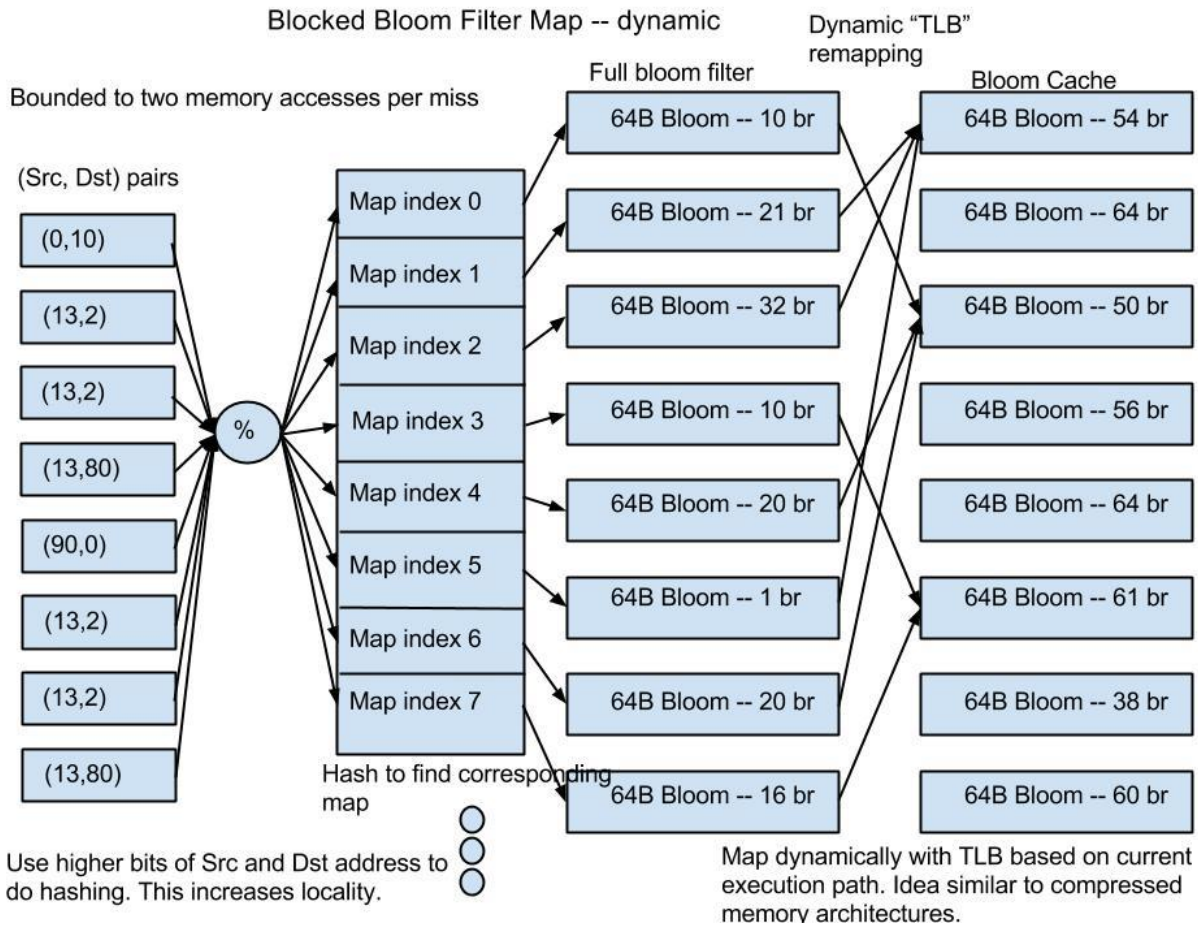

Figure 5: Dynamic Bloom Cache

I introduce a dynamic Bloom cache to incorporate temporal locality dynamically in an unbalanced blocked Bloom filter design. Based on the property that two Bloom filter blocks can be combined with a simple bitwise OR, I combine sparse blocks in the cache structure. The full bloom filter is kept sparse, but the Bloom Cache combines blocks temporarily to better utilize the cache space alotted. This allows the Bloom cache to have multiple temporal contexts in one Bloom block and to ideally have 64 branches in every

64B block cached despite the full Bloom filter being unbalanced. To maintain the mapping of indices to corresponding merged blocks, I propose to either use a structure similar to a variable-segment cache [38] or a highly associative CAM structure similar to a TLB. The idea of a variable-segment cache is that one can increase effective memory storage by storing multiple compressed blocks into a single physical block. To implement this, one uses multiple tags per indexed block.

This dynamic Bloom allows for both spatial locality and temporal locality, but it also comes with other interesting properties. Since the blocks are combined during cache Bloom insertion, one can additionally dynamically control the rate of false positives, by choosing the level of Bloom filter fill. For example, if the program is executing system calls or potentially vulnerable library calls, one could set the maximum fill from 64 branches to 32 branches per 64B and subsequently reduce the false positive rate from 2% to 0.1% [25], increasing the level of security at the cost of some performance loss due to the halved effective cache capacity. Conversely, if code is known to be relatively safe, one can fit, for example, 100 branches in a block for a false positive rate of 10%. Alternatively, one could even potentially profile hit rate and automatically increase Bloom Cache fill ratio to increase hit rate when the workload is memory intensive or lower fill rate to keep a high level of security in the general case when memory is not in heavy use.

Specific Implementation Issues

Since the blocked Bloom filter is unbalanced, the full Bloom filter itself can be sparse, as there are many indices (and corresponding blocks) with 0 branches inside. To allocate 64B regardless would make all accesses quick, but the amount of space required would be too large per program. One could create a map, similar to a page table, to find the corresponding location, but even approach that has its limits. To handle the empty blocks from an unbalanced blocked Bloom filter, I use a technique from [29] called

"Cuckoo Hashing." Cuckoo Hashing is similar to a hash table in that the index is hashed to find the proper storage location. However, Cuckoo Hashing works by having two independent hashes and allowing each block to be in one of the two places calculated by the two independent hashes. When an insertion has a collision with a previous block, other blocks are shifted around their two valid positions until all the blocks fit. This results in a fill rate of about 50%.

An example of fill rate and size requirements for a full blocked Bloom filter is shown here. This example shows storage requirements of bzip2 benchmark. For spatial locality, this configuration ignores 10 least-significant bits of source and destination addresses for locality and uses 10 higher-order bits of source and destination addresses to index the block. In this particular configuration, there are 8 tags per Bloom index to allow for dynamic combination of blocks in the Dynamic Bloom Cache.

Table 1: Example memory usage of a Dynamic Bloom Cache

Number of branches: 1392 Number of mappings: 364
Max Fill: 65 Average Fill: 3.824176 Stddev: 169.099814
Bloom Filter Size: 2048 B
Tagging overhead: 800 B
Full Map size – dynamic bloom without hash map – single access: 65536 kB
Full Map size – dynamic bloom with cuckoo hash: 45 kB
Full Map size – dynamic bloom minimum space: 22 kB
Full Map size – single Bloomfilter minimum space: 1 kB

In general, the average fill is not high (3.8 out of 64 that can be stored to maintain a 2% false positive), so the space required in disk / main memory (map size) may be higher than a single Bloom filter. However, storing Bloom filters in the blocked format reduces the amount of quickly-accessible cache space needed, as blocks can be loaded and evicted as needed. This particular design needs 2048B of Bloom filter and 800B for tags, as opposed to the 32kB requested in [10]. As access patterns in general program

execution exhibit spatially and temporally local branch accesses, the hit rate of such a filter is high.

CHAPTER 5

IMPLEMENTATION

To model overhead, I have used MacSim [30], a trace-based heterogeneous architecture timing model simulator that is based on a Pin front-end [31]. MacSim has been shown to be very accurate and efficient and papers using MacSim have been accepted and presented in renowned conferences such as ISCA. Pin trace-generation is supported by Intel and has been shown to be very efficient with many recent papers have been using pin in their timing simulations.

To model the delay, I perform a branch-verification on every branch target miss, as branch target hits are branches that have been seen and thus verified before. As the Bloom hashing and accessing timings were shown to be less than branch misprediction penalty in [Shi] and can thus be masked, the Bloom hashing is modeled as zero additional delay. The performance overhead is modeled by stalling the execution on a Bloom block mapping miss, as the Bloom Cache must load the relevant Bloom filter block from memory into cache.

The CPU modeled is a 4-instruction-wide single-core x86 architecture with out-of-order micro-op timing simulation to more accurately represent commercially available computers. A branch target buffer is used to predict branch targets. Memory parameters include 64kB L1 cache, 256kB L2 cache, 1 MB L3 cache, and 64B line sizes.

CHAPTER 6

PERFORMANCE CHARACTERIZATION

The proposed Bloom filters will be compared by running timing simulations on 1 billion representative instructions picked by SimPoint [32] [33] from the Spec2006 [34] benchmarks. The full Bloom filters are populated by collecting a branch trace from start-to-end execution of the benchmarks. Overall, the Bloom Cache seems to perform reasonably well as it maintains Bloom block mapping hit rates of > 98% despite using cache space significantly smaller than the number of branches in the full Bloom filter.

To show the overhead of the proposed research, this chapter will simulate various granularities of branch verification and show general performance overheads of the different approaches. The chapter will start by start by simulating the overhead for verifying both indirect and direct branches by using a full bloom filter trained from the set of 1 billion instructions (6a). This simulation is intended to compare the overhead in using the different proposed configurations. Subsequent sections use full bloom filters that store indirect branches trained from entire program execution. The next section (6b) models overhead from checking only indirect branches. Checking only indirect branches for verification is based on the notion that direct conditional branches and direct calls have set targets, so direct branches are not vulnerable to attack as long as the stack is protected by DEP. As for not checking return instructions, return instructions can be protected with a shadow call stack[1]. Following that, in section (6c), use of additional dynamic branch signatures to improve security are examined.

Check All Branches

As shown in N-jump [35] suitable protection can be achieved by verifying all branches. The initial set of experiments is intended as a quick measure of performance

characterization between the different approaches. The Bloom filter is also filled with both direct and indirect branches seen in the 1 billion instruction trace.

Run time: (Run time with branch verification / Run time without branch verification)

Table 2: Performance overhead from checking direct and indirect branches

	# branches	Static ideal	Static Hierarchical	Dynamic 4-way 8x tag	Dynamic fully assoc. 8x tag
Astar	638	1.000003	1.000003	1.000009625	1.000009625
bzip2	1355	0.999994	0.999994	1.000004994	1.000004671
Gcc	7855	1.027268	1.051634	1.090702527	1.072012034
Gobmk	10259	1.081345	1.169818	1.055456873	1.025391371
Hmmer	862	1.000000	1.000000	1.00002276	1.000123803
Mcf	108	1.000011	1.000011	0.999898896	0.999898896
perlbench	11023	1.118754	0.88112*	0.86273851*	0.869153918*
Sjeng	1616	1.000273	1.000331	1.000579725	1.000003715
Avg	3242	1.015556	1.031685	1.020953629	1.013920588

*perlbench results give odd results at times. A possible explanation is that the assumed location of the full bloom filter location is memory actively used by the benchmark. Average overheads do not include perlbench.

It can be seen that in most cases, because the number of valid branches to check are small, most of the Bloom filter can fit into the Bloom cache and thus almost always have the Bloom mappings hit. This causes overhead in most of the cases, namely, the programs with < 2000 valid branches, to be negligible. However, in the case of gcc, gobmk, and perlbench, there are more valid branches, and we can see slowdown resulting from accessing the full Bloom filter. It can be seen that the static configurations that only consider of spatial locality do not scale as the number of branches increase, as the performance degrades significantly once the number of valid branches is over 8000. As

for dynamic configurations that account for temporal locality, the performance degradation is bound instead by 7%, and benchmarks with more branches do not necessarily perform worse as their branch patterns could have more temporal locality and hit in the Bloom cache more often. In the case of gcc, the static configurations actually outperform the dynamic configurations: this is because gcc is more memory-intensive, and blocked Bloom configurations with larger full Bloom filters on disk could take up more space in L2 or L3 caches, weakening the verification performance. A way to avoid using cache space that memory-intensive applications could be to increase the Bloom cache size and read all of the blocks directly from memory without caching in L2 or L3 caches. For this to be efficient, the Bloom cache map must have a very high hit rate.

I have shown here some example dynamic Bloom cache hit rates with respect to the set-associativity of the cache and the number of tags per Bloom map index. These hit rates are calculated with respect to gcc benchmark, which has 7855 valid branches that would not fit in a 2kB cache. The 2kB cache at maximum capacity, with 64 branches in every 64B bloom block in the bloom cache, can only store 2000 branches at a time. However, due to the ability of the proposed dynamic bloom cache to make use of spatial and temporal locality, the hit rates of the blocked bloom filters are often >97%.

Replacement policies used: Insert lowest index available. LRU replacement.

Table 3: Dynamic Bloom Cache hit rates for different configurations

2k bloom (10bits locality, 10bits length)	4x tags	8x tags	16x tags
4-way	0.939268	0.972968	0.974493
8-way	0.942910	0.977540	0.978780
16-way	0.945344	0.980631	0.981734
fully associative	0.946806	0.983017	0.983416

It can be seen that the hit rate receives the most benefit from having 8x the number of tags as the number of physical Bloom mappings, so subsequent configurations assume 8x the number of tags as physical indexes.

Check All Indirect Branches

From [10], it can be seen that the most vulnerable of branches are indirect branches and returns, as an attacker can change the value in a register to potentially jump to any location he wants. To protect against code re-use attacks, verifying all of the indirect branches was said to be sufficient. The valid indirect branches are obtained by running a program to completion.

The overhead in checking all indirect branches are as follows:

Table 4: Performance overhead from checking indirect branches

Check indirect branches	# indirect branches and returns	Dynamic Bloom execution time (check all indirect branches)	Dynamic Bloom execution time (check indirect branches except ret)
Astar	983	1.032605780	0.999996226
bzip2	1392	1.058523978	0.999996386
Gcc	19241	1.004170487	1.001347175
Gobmk	7483	1.164263847	0.997408238
Hmmer	1624	1.000721573	1.002443695
Mcf	487	1.025255816	1.000000000
Perlbench	13700	0.877273072*	1.004178741
Sjeng	1028	1.116015178	0.999779594
Average	5742	1.057365	1.000139

*perlbench results are not included in the average.

The average performance overhead in checking all indirect branches is 5.7% while not checking returns gives performance overhead of .01%. This shows that the overhead in checking returns are modest as returns can return to many potential call sites. The return checking overhead can be reduced by checking on return-address-stack misses, but this has not been simulated yet.

Different Branch Signatures

In addition to keeping track of (src, dst) pairs, one could provide additional security by checking against a dynamic execution path. This would prevent users from using valid branches in a potentially abusive order. One could add information from past branch direction histories as in a branch history register (BHR), called Execution Path in [10], or one could be more specific and have a history of calling site by storing the LSB of the previous branch pc in a Path History Register (PHR). Used in combination, this can verify that the previous N branches were the same and that those previous N branches executed in the same direction as before.

The following table shows the number of indirect branch signatures (src||dst||bhr||phr) for start-to-end runs of benchmark applications:

Table 5: Number of (src||dst||bhr||phr) branch signatures in each benchmark

2k bloom	0-bit phr+bhr	1-bit	2-bit	4-bit	8-bit
astar	983	983	1090	1307	2189
bzip2	1392	1392	1532	1782	2680
gcc	19241	19241	22850	32846	61181
gobmk	7483	7483	9061	13163	28664
hmmer	1624	1624	1740	2137	3190
mcf	487	487	530	611	867
perlbench	13700	13700	14812	25577	54844

Table 5 continued

sjeng	1028	1028	1226	1927	5221

For most applications, the behavior signatures do not increase much when more dynamic path verification is added because paths taken just before indirect branches seem to be consistent up to about 4 paths. However, when one takes branch histories of 8 previous paths, most applications see a significant jump in number of signatures, meaning that the many possible control flows preceding the current control flow is too erratic. Using too long of a history would decrease performance and require longer training periods, so one should limit the use BHR and PHR to about 4 bits.

CHAPTER 7

EXAMPLE ATTACK – BUFFER OVERFLOW

An example buffer overflow attack is shown here to show the efficacy of the proposed bloom filter design [12].

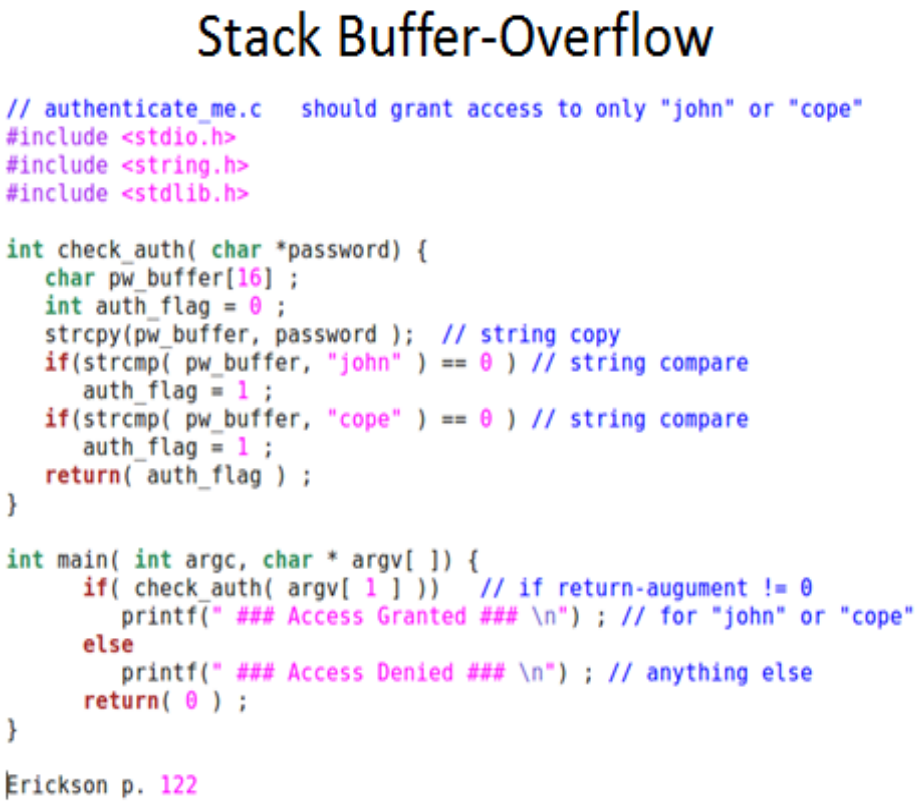

Figure 6: Buffer overflow example code

This code base shows a standard buffer overflow potential, where a command line input can be larger than the buffer allocated to read the input.

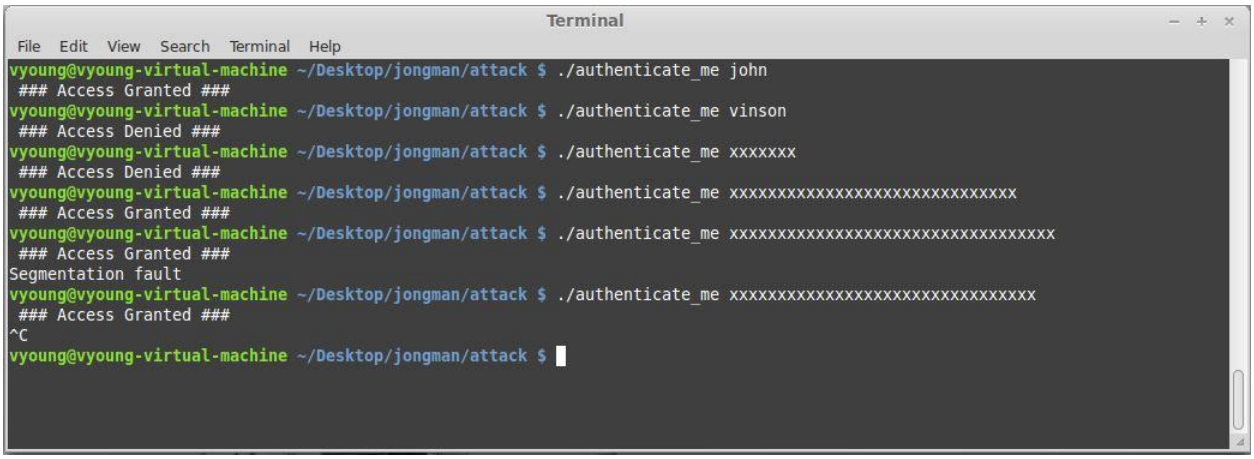

Figure 7: Buffer overflow in action

With such a small input buffer, one can overwrite the verification flag with a sufficiently long input string, as denoted by the input string of 30 x's. This constitutes a data-only attack, and is not detected by the branch verification proposed. However, if one supplies an even longer input string, one can overwrite the value in the potentially vulnerable return instruction and branch to an arbitrary location. This is potentially a control flow hijacking attempt as it could reroute execution flow to code specified by the attacker. This will be caught by Control Flow Integrity implementations as the aberrant return would be checked against valid branches and be found not to be valid, unless it happened to lie within the small false positive rate or branches to other valid control points.

CHAPTER 8

FURTHER CONSIDERATIONS

Further considerations for the proposed Bloom blocked design include (1) dynamically changing protection levels, (2) buffering checks, and (3) adapting bloom cache for general use. (1) The level of protection in the dynamic Bloom cache design can be changed as needed for performance or for security. (2)The bit verifications can be stored in a buffer to wait for the Bloom map to access the blocked Bloom from main memory. This can delay checks and would reduce the overhead as there could be multiple outstanding verifications at a time. As purported in [28], the most egregious of control flow hijacking attacks stem from hijacking system calls, so one can delay general calls for more efficient checking. (3) This Bloom Cache could potentially be used in other applications where there is spatial and temporal locality in queries. Bloom filters are currently used in database lookups of Google BigTable and Apache Cassandra [36] and malicious url checks of Google Chrome [37]. These applications could have spatial and temporal locality of access and may be improved. Specifically, this could be used to implement a firewall / malicious url checker on servers if requests are sufficiently local. The Bloom Cache could also be used for verifying certificate revocation lists quickly, assuming locality of access. Another possible application would be to check memory accesses and make sure no hacker maliciously makes changes immutable kernel regions. Further investigation of potential uses outside of branch verification will be explored in the future.

CHAPTER 9

CONCLUSION

In this work, I have researched and found that code re-use attacks are not sufficiently protected against in commodity solutions and that current implementations of solutions proposed in research, like CFI, either incur large performance overheads, weaken code compatibility, or are unscalable. To overcome performance and code compatibility issues, I use a Bloom-filter-based approach. To overcome scalability issues of a single-Bloom-filter implementation, I propose several new methods to add spatial and temporal locality to a blocked bloom filter approach to reduce the amount of quickly-accessible cache space that must be used to maintain performance. The solution is also shown to be scalable as there is not necessarily a positive correlation between number of branches stored and performance overhead. The average performance overhead of using the proposed dynamic bloom cache to check all indirect branches is 5.7%, this being a conservative estimate as a Return Address Stack was not modeled. The overhead is an improvement to the 22% of the original CFI paper [1], and this configuration additionally solves code modularity issues. Using the proposed bloom cache structure also addresses the scalability issues of the configuration as proposed in the single bloom-filter architecture proposed by [10]. In summary, this work proposes a high-performing, scalable and stateless general security solution to defend against control flow hijacking attempts.

REFERENCES

[1] Martín Abadi , Mihai Budiu , Úlfar Erlingsson , Jay Ligatti, Control-flow integrity principles, implementations, and applications, ACM Transactions on Information and System Security (TISSEC), v.13 n.1, p.1-40, October 2009.

[2] "Data Execution Prevention." Resources and Tools for IT Professionals | TechNet. N.p., n.d. Web. 2 Oct. 2013. <http://technet.microsoft.com/en-us/library/cc738483(v=ws.10).aspx>.

[3] Tran, Minh, et al. "On the expressiveness of return-into-libc attacks." Recent Advances in Intrusion Detection. Springer Berlin Heidelberg, 2011.

[4] Roemer, Ryan, et al. "Return-oriented programming: Systems, languages, and applications." ACM Transactions on Information and System Security (TISSEC) 15.1 (2012): 2.

[5] Bletsch, Tyler, et al. "Jump-oriented programming: a new class of code-reuse attack." Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. ACM, 2011.

[6] Team, PaX. "PaX address space layout randomization (ASLR)." (2003).

[7] Tyler Durden. Bypassing PaX ASLR Protection. Phrack Magazine, Volume 11, Issue 0x59, File 9 of 18, June 2002.

[8] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address Space Randomization. 11th ACM CCS, 2004.

[9] Burton Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of ACM, pages 13(7):422-426, July 1970.

[10] Y. Shi and G. Lee, "Augmenting branch predictor to secure program execution," in Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on. IEEE, 2007, pp. 10-19.

[11] Kanter, David. "Intel's Sandy Bridge Microarchitecture." Real World Tech. Real World Tech, 25 Sept. 2010. Web. 6 Apr. 2014. <http://www.realworldtech.com/sandy-bridge/4/>.

[12] Copeland, John. "6612/slides." ECE 6612 Computer Network Security (2S, 1D) 3-0-3. Communications Systems Center at Georgia Institute of Technology, 1 Apr. 2014. Web. 6 Apr. 2014. <http://www.csc.gatech.edu/copeland/jac/6612/slides/index.html>.

[13] Sandhu, Ravi S., et al. "Role-Based Access Control Models yz." IEEE computer 29.2 (1996): 38-47.

[14] "TrustZone - ARM." ARM - The Architecture For The Digital World. N.p., n.d. Web. 2 Oct. 2013.

[15] Gerald J. Popek and Robert P. Goldberg (1974). "Formal Requirements for Virtualizable Third Generation Architectures". Communications of the ACM 17 (7): 412 –421.

[16] Wang, Zhi, and Xuxian Jiang. "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity." Security and Privacy (SP), 2010 IEEE Symposium on. IEEE, 2010.

[17] Xia, Yubin, Yutao Liu, and Haibo Chen. "Architecture support for guest-transparent VM protection from untrusted hypervisor and physical attacks." High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on. IEEE, 2013.

[18] Wang, Jiang, Angelos Stavrou, and Anup Ghosh. "HyperCheck: A hardware-assisted integrity monitor." Recent Advances in Intrusion Detection. Springer Berlin Heidelberg, 2010.

[19] Szekeres, L.; Payer, M.; Tao Wei; Song, D. "SoK: Eternal War in Memory", Security and Privacy (SP), 2013 IEEE Symposium on, On page(s): 48 – 62

[20] Castro, Miguel, Manuel Costa, and Tim Harris. "Securing software by enforcing data-flow integrity." Proceedings of the 7th symposium on Operating systems design and implementation. USENIX Association, 2006.

[21] Lam, Lap Chung, and Tzi-cker Chiueh. "A general dynamic information flow tracking framework for security applications." Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual. IEEE, 2006.

[22] Baliga, Arati, Vinod Ganapathy, and Liviu Iftode. "Detecting kernel-level rootkits using data structure invariants." Dependable and Secure Computing, IEEE Transactions on 8.5 (2011): 670-684.

[23] Cowan, Crispin, et al. "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks." Proceedings of the 7th USENIX Security Symposium. Vol. 81. 1998.

[24] Cowan, Crispin, et al. "Pointguard TM: protecting pointers from buffer overflow vulnerabilities." Proceedings of the 12th conference on USENIX Security Symposium. Vol. 12. 2003.

[25] Cao, Pei. "Bloom Filters - the math." Bloom Filters - the math. University of Wisconsin, 5 July 1998. Web. 6 Apr. 2014. <http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html>.

[26] Tyler Bletsch , Xuxian Jiang , Vince Freeh, Mitigating code-reuse attacks with control-flow locking, Proceedings of the 27th Annual Computer Security Applications Conference, December 05-09, 2011, Orlando, Florida

[27] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev. Branch regulation: Low overhead mitigation of code reuse attacks. In Proceedings of ISCA, 2012.

[28] Linbo Chen; Jianhui Jiang; Danqing Zhang, "Code Reuse Prevention through Control Flow Lazily Check," Dependable Computing (PRDC), 2012 IEEE 18th Pacific Rim International Symposium on , vol., no., pp.51,60, 18-19 Nov. 2012.

[29] Pagh, Rasmus, and Flemming Friche Rodler. "Cuckoo hashing." Journal of Algorithms 51.2 (2004): 122-144.

[30] Kim, Hyesoon, et al. "MacSim: A CPU-GPU Heterogeneous Simulation Framework User Guide." Georgia Institute of Technology (2012).

[31] Luk, Chi-Keung, et al. "Pin: building customized program analysis tools with dynamic instrumentation." ACM Sigplan Notices 40.6 (2005): 190-200.

[32] Perelman, Erez, et al. "Using SimPoint for accurate and efficient simulation." ACM SIGMETRICS Performance Evaluation Review. Vol. 31. No. 1. ACM, 2003.

[33] Timothy Sherwood, Erez Perelman, Greg Hamerly and Brad Calder. Automatically Characterizing Large Scale Program Behavior, In the proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2002), October 2002. San Jose, California.

[34] Henning, John L. "SPEC CPU2006 benchmark descriptions." ACM SIGARCH Computer Architecture News 34.4 (2006): 1-17.

[35] T. Zhang, X. Zhuang, W. Lee, S. Pande, "Anomalous Path Detection with Hardware Support," in Proc. of CASES, 2005.

[36] Chang, Fay; Dean, Jeffrey; Ghemawat, Sanjay; Hsieh, Wilson; Wallach, Deborah; Burrows, Mike; Chandra, Tushar; Fikes, Andrew; Gruber, Robert (2006), "Bigtable: A Distributed Storage System for Structured Data", Seventh Symposium on Operating System Design and Implementation.

[37] Yakunin, Alex. "Nice Bloom filter application." Alex Yakunin's blog. N.p., 25 Mar. 2010. Web. 6 Apr. 2014. <http://blog.alexyakunin.com/2010/03/nice-bloom-filter-application.html>.

[38] Alameldeen, Alaa R., and David A. Wood. "Adaptive cache compression for high-performance processors." Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on. IEEE, 2004.